Real-Time Analysis of a HW-SW Platform

朱致伶, 110550062

***Abstract*—In this homework, I added profiling hardware to the Aquila core to analyze the CoreMark benchmark program execution behavior. By implementing a self-design profiling mechanism, I was able to count the total CPU cycles and compute the CPU ratio of the top 5 hotspots of CoreMark benchmark program, comparing the result with gprof, which is a software profiler. Also, I recorded the ratio of computation versus memory cycles and analyzed the stall cycles distribution.**

***Keywords*—aquila; CoreMark; hotspot; profiler; computation; memory; stall (key words)**

I. INTRODUCTION

Real-time analysis of a hardware-software platform refers to the process of continuously monitoring and analyzing the performance and behavior of components in real-time, which is an important performance analysis tool nowadays. Profiler does real-time analysis of specific programs, which can be divided into software and hardware profiler. Since hardware profiler has less limitations and more precise statistics, I compared the profiling result between hardware self-design profiling mechanism running on Aquila with software profiler gprof.

In this homework, I took CoreMark benchmark program as target application. To profile the CoreMark program, I computed the CPU ratio of top 5 hotspot functions in gprof results, the computation versus memory access cycles counts. In addition, I observed and analyzed the stall cycle counts of each function caused by different reasons. Lastly, I made a conclusion from the previous results and proposed some suggestions to improve Aquila.

II. PROFILING MECHANISM

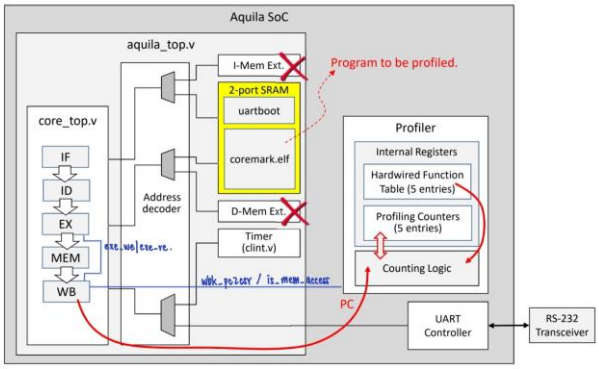

Fig. 1. Modified Aquila SoC with Profiling Mechanism.

The profiling mechanism in Aquila can be divided into the following two functions:

A. CPU ratio of top 5 hotspot functions

To compute the CPU ratio, I had to check whether the current instruction belongs to the hotspot function, then create counters to record the value of total clock cycles and the cycles of each hotspot function. First, I connected the **wbk_pc2csr** wire from writeback stage to profiler since instructions after writeback stage must be implemented successfully. To check whether the current instruction belongs to any hotspot function, I found the hotspot functions' fixed start and end addresses in the coremark.objdump file, then set the interval of the two addresses as the counter condition. As for computing the total clock cycles, I triggered the start signal of total clock cycles count when the main function of CoreMark is called and stopped counting as the main function return.

Knowing the precise start and end address of the main and top 5 hotspot functions, the profiler is able to compute the CPU ratio of the hotspot function by dividing the cycle count of each function with cycle count of main. Table. 1 shows the start and end address of main and hotspot functions:

TABLE I. START / END ADDRESS OF PROFILING FUNCTIONS

Function Name	***Start Address***	***End Address***
Main	32'h1088	32'h1794
core_list_find()	32'h1d28	32'h1d7c
core_list_reverse()	32'h1d80	32'h1da0
core_state_transition()	32'h2a14	32'h2d0c
matrix_mul_matrix_bitextract()	32'h2670	32'h272c
crcu8()	32'h19e8	32'h1a28

[a] *Start and End Address are displayed in hexadecimal radix /*

B. Ratio of Computation versus Memory Access Cycles

The difference between computation and memory cycle is whether it accesses memory as implementing the current instructions. To determine whether the instruction accesses the memory, I connected the write and read enable wire from execution stage to writeback stage. The reason I didn't catch the read and write enable signals from instruction decode stage is that the instructions may encounter flush or exception during the register between instruction decode and execution, catching the signals from execution stage can effectively reduce these cases.

However, there is a potential problem since the write and read signals skip the memory stage, causing the instruction passed to the writeback stage isn't the one with read and write enable signals. The solution to the problem is to create the **is_mem_reg** registers in writeback stage storing whether the instruction corresponds to the read and write enable signals in execution stage access memory in memory stage. In detail, there are four cases in writeback stage listed in Table. 2:

TABLE II. CASES OF INSTRUCTIONS IN EXE_STAGE ACCESS MEMORY

	Writeback Stage	
	condition	***is_mem_reg***
Case 1	rst_i || (flush_i && !stall_i)	1'b0
Case 2	stall_i	is_mem_reg
Case 3	xcpt_valid_i	1'b0
Case 4	Other cases	exe_we | exe_re

After knowing whether the current instruction accesses memory in writeback stage in profiler, combine the previous if-else statement, the profiler can calculate the ratio of computation versus memory cycles for each function.

III. EXPERIMENTAL RESULTS AND DISCUSSION

Table. 3 shows the cycle count and ratio of the top 5 hotspot functions of CoreMark and cycle counts of computations and memory access cycles for each function. The top 5 hotspot functions of CoreMark application are core_list_find(CLF), core_list_reverse(CLR), core_state_transition(CST), matrix_mul_matrix_bitextract(MMMB) and crcu8(CRCU8).

TABLE III. CYCLE COUNT AND RATION OF HOTSPOT FUNCTIONS

Cycle Types	**Function Name**				
	CLF	***CLR***	***CST***	***MMMB***	***CRCU8***
Total	79614376	53546130	113536720	104248760	68562102
Computation	58415176	31330530	97216240	93261960	68562102
Memory	21199200	22215600	16320480	10986800	0
Ratio	12.95%	8.71%	18.46%	16.95%	11.15%

Total Cycle Count = 614878275, Total Computation Cycle = 348786008, Total Memory Cycle = 70722080

A. CPU ratio of top 5 hotspot functions

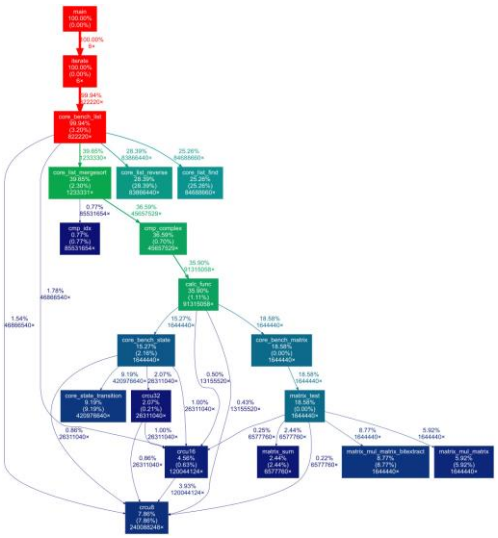

Fig. 2. CoreMark Graph Profiled with Software Profiler gprof

Table 4 listed the CPU ratio of each function profiled by software and hardware profiler. Compared to the statistics, there is no clear relation between the software and hardware profiler CPU ratio. These are the potential reasons behind the results:

- Software profiler's timer interruptions can be obstructive to the program.
- The DIteration variable in coremark.elf file is set to 0, which means that the program automatically stops as it determines the cycles of functions converge, which is an uncertainty.

TABLE IV. CYCLE COUNT AND RATION OF HOTSPOT FUNCTION

SW/HW Profiler	**Function Name**				
	CLF	***CLR***	***CST***	***MMMB***	***CRCU8***
gprof	25.26%	28.39%	9.19%	8.77%	7.86%
Profiling Mechanism	12.95%	8.71%	18.46%	16.95%	11.15%

B. Ratio of Computation versus Memory Access Cycle

From Table III, we know that the top 5 hotspot functions implement more computation instructions than memory access instructions.

Take a deeper look at each function respectively, core_list_find function finds elements in list and core_list_reverse function reverses the list; therefore, it is easy to understand it has more computation cycles than memory ones. As for core_state_transition function, an actual state machine, and matrix_mul_matrix_bitextract function, which is responsible for matrix computation, both functions have much more computation cycles than memory. Since crcu8 serves as a part of calculating 16bits CRC code, it does no memory access cycles.

C. Stall Cycles of Each Function

Table.V shows the stalls cycle count caused by different reasons of each function. With tracing the following code, I got to analyze the reason of causing stall_data_fetch and stall_data_hazard:

```
assign stall_instr_fetch = (!code_ready_i);
assign stall_data_fetch = (dS_nxt == d_WAIT) && (! exe_is_fencei);
assign stall_pipeline = stall_instr_fetch | stall_data_fetch | stall_from_exe;
```

TABLE V. STALL CYCLE COUNT OF HOTSPOT FUNCTIONS IN PROFILER

Cycle Types	**Function Name**				
	CLF	***CLR***	***CST***	***MMMB***	***CRCU8***
total_stall	20949940	14810400	1349320	55766440	0
data_fetch	20949940	14810400	1349320	7458400	0
from_exe	0	0	0	48308040	0
data_hazard	20825310	0	4026880	0	0

1) Stall_data_fetch

The main reason for causing stall_data_fetch is the statement

ds_nxt == d.WAIT

of finite state machine in core_top.v. By tracing the code, it finally stopped at the below code in decode.v:

assign re = rv32.load | rv32.amo
assign we = rv32.store | rv32.fencei

Stall happens when decode.v keep passing new instructions from execution to memory stage. As a conclusion, when current instructions implement load or store instructions, the stall_data_fetch cost 1 cycle to access memory, which is inevitable.

IV. DISSCUSSION TO IMPROVE AQUILA

In this homework, I observed how Aquila handle stalls situation, and thought this may be an entry point to improve the Aquila performance by pipeline optimization. To improve the Aquila SoC by decreasing stall cycles, the followings are the future development direction to try:

- Pipeline Optimization: By optimizing the pipeline design, you can reduce stalls and improve overall performance. Techniques such as instruction scheduling, branch prediction, and data forwarding can help minimize stalls caused by dependencies and branch mispredictions.
- Instruction-Level Parallelism: Explore opportunities for instruction-level parallelism within the Aquila SoC. Techniques such as instruction pipelining, superscalar execution, and out-of-order execution can help overlap and execute multiple instructions simultaneously, reducing stalls caused by dependencies.